Synchronous FPNNs: neural models that fit reconfigurable hardware

Bernard Girau

Synchronous FPNNs: neural models that fit reconfigurable hardware

Bernard Girau

Abstract

Neural networks are considered as naturally parallel computing models. But the number of operators and the complex connection graph of standard neural models can not be handled by digital hardware devices. Neural network hardware implementations have to reconcile simple hardware topologies with complex neural architectures. The theoretical and practical framework developped in [16] allows this combination by means of some configurable hardware principles applied to neural computation: *Field Programmable Neural Arrays* (FPNA) lead to powerful neural architectures that are easy to map onto FPGAs, thanks to a simplified topology and an original data exchange scheme, without having any significant loss of approximation capability. This report follows the overview of FPNAs in report [17]: it focuses on a family of FPNA-based neural networks that are more specially adapted to *re*configurable hardware.

1 Introduction

Configurable hardware devices such as FPGAs (field programmable gate arrays) are cheap, flexible, and they offer both digital hardware efficiency and simple software-like handling. Moreover reprogrammable FPGAs allow neural network prototyping, and FPGA-based implementations of neural networks may be mapped onto new improved FPGAs (unlike the use of neuroprocessors, which rapidly become outdated). Therefore configurable hardware, and configurable computing systems by extension, appear as well-adapted to obtain efficient and flexible neural network implementations. Conversely, neural computation features, such as massive fine-grain parallelism and locality, show that neural network applications are obvious candidates to take advantage of the emerging technology of configurable computing systems.

However, the 2D-topology of FPGAs does not allow to handle the *connection complexity* of standard neural network models. Moreover, FPGAs still implement a limited number of logic gates, whereas neural computations (multipliers, transfer functions) are *area-greedy* operators. Usual solutions ([5, 11, 24, 8, 7, 10, 4]) handle sequentialized computations with a FPGA used as a small neuroprocessor, or they implement very small low-precision neural networks without on-chip learning. Connectivity problems are not solved even by the use of several reconfigurable FPGAs with a bit-serial arithmetic ([9]), or by the use of small-area operators (stochastic bitstream in [2], frequency-based in [20]).

In [18, 19], we propose an implementation method for multilayer perceptrons of any size on a single FPGA, with on-chip learning and adaptable precision. This work takes advantage of an area-saving on-line arithmetic that is well-adapted to neural computations. It also uses an original parallelization of the internal computations of each neuron. Yet, this implementation method barely exploits the parallelism induced by neural network architectures.

The work described in [16] aims at developping *neural architectures* that are easy to *map onto FPGAs*, thanks to a *simplified topology* and an original *data exchange scheme*, without having any significant loss of approximation capability. It has been achieved thanks to the definition

of a set of neural models called *Field Programmable Neural Arrays* (FPNA). FPNAs may lead to the definition of neural networks adapted to hardware topological constraints. Different such neural networks may be derived from a given FPNA. They are called *Field Programmed Neural Network* (FPNN). They reconcile the high connection density of neural architectures with the need of a limited interconnection scheme in hardware implementations.

Report [17] proposes a brief overview of FPNA and FPNN definitions, computations, and implementations. The present report first recalls the general FPNA concept. Then it focuses on synchronous FPNNs, which characteristics may be more fully exploited thanks to *re*configurable hardware. Readers of report [17] may consult this one from section 3.

2 FPNAs, FPNNs

The distinction between FPNAs and FPNNs is mainly linked to implementation properties. A FPNA corresponds to a given set of neural resources that are organized according to specific neighborhood relations. A FPNN is a way to use this set of resources: it only induces local configuration choices. Therefore, the implementation needs of two different FPNNs are the same, as long as they are based on the same FPNA.

2.1 FPNAs

2.1.1 From FPGAs to FPNAs

The first aim of the FPNA concept is to develop neural structures that are easy to map onto digital hardware, thanks to a simplified and very flexible topology. Therefore, the structure of a FPNA derives from FPGA principles (complex functions realized by means of a set of simple programmable resources), while the nature and the relations of FPNA resources derive from the mathematical processing FPNAs have to perform. These resources are communication links and computation elements.

To summarize, in a standard neural model, each neuron computes a function applied to a weighted sum of its inputs: if $\vec{x_i}$ is the input vector of neuron i, and $\vec{w_i}$ is its weight vector, it computes $f_i(\vec{w_i}.\vec{x_i})$. See [12, 13, 14] for a unified theoretical approach of the computation of standard neural networks. The input vector $\vec{x_i}$ may contain neural network inputs or outputs of other neurons, depending of the *connection graph* of the neural network. In such a standard model, each communication link is a connection between the output of a neuron and an input of another neuron. Therefore the number of inputs of each neuron is its fan-in in the connection graph. On the contrary, communication links and neurons become *autonomous* in a FPNA: their dependencies are freely programmable, and may result in a more complex processing than in standard neural models.

2.1.2 FPNA resources

FPNAs are defined to compute partial convolutions for non-linear regression ([15]), as standard multilayer neural networks do. Nevertheless their architecture is simplified with respect to standard neural models.

A FPNA is intended to use programmable hardware principles to allow direct mappings of various neural networks onto digital hardware. Thus it is made of a programmable set of neural resources. Two kinds of autonomous FPNA resources naturally appear: *neurons* that apply standard neural functions to a set of input values on one hand, and *communication links* that behave as independent affine operators on the other hand.

These resources may be handled in different ways. The easiest scheme would allow to allocate any communication link to any neuron, with the help of an underlying programmable interconnection net. This would lead to massively pruned standard neural networks, or to multiple weight sharing connections. Topological problems may still appear (such as high fan-ins). And weight sharing would induce few different $\vec{w_i}$ weight vectors. Therefore, the FPNA concept allows to *connect* any communication link to any *local resource*. The aim of locality is to reduce topological problems, whereas connected communication links result in more various weight vectors.

More precisely, the communication links connect the nodes of a directed graph, each node contains one neuron. The specificity of FPNAs is that relations between *any* of the local resources of each node may be freely set. A link may be connected or not to the local neuron *and to the other local links*. Direct connections between affine links appear, so that the FPNA may compute numerous composite affine transforms. These compositions create numerous *virtual neural connections*, so that different convolution terms may be obtained with a reduced number of connection weights.

2.1.3 Formal definition of FPNAs

A FPNA is defined by means of:

- A directed graph $(\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ is a ordered finite set of nodes, and $\mathcal{E}$ is a set of directed edges ($\mathcal{E}$ can be seen as a subset of $\mathcal{N}^2$).

  For each node n, the set of the direct predecessors (resp. successors) of n is defined by $Pred(n) = \{p \in \mathcal{N} \mid (p,n) \in \mathcal{E}\}$ (resp. $Succ(n) = \{s \in \mathcal{N} \mid (n,s) \in \mathcal{E}\}$). The set of the input nodes is $\mathcal{N}_i = \{n \in \mathcal{N} \mid Pred(n) = \emptyset\}$.

- A set of affine operators $\alpha_{(p,n)}$ for each (p,n) in $\mathcal{E}$.

- A set of "neurons" (i_n, f_n), for each n in $\mathcal{N} - \mathcal{N}_\rangle$: i_n is an iteration operator (a function from $\mathbb{R}^2$ to $\mathbb{R}$), and f_n is a transfer function (from $\mathbb{R}$ to $\mathbb{R}$).

2.1.4 Interpretation

Resources are associated with the nodes, whereas locality is defined by the edges. For each node $n \in \mathcal{N}$, there is one neuron resource and as many communication links as this node has got predecessors. Each communication link is associated with an affine operator. A neuron resource is defined by (θ_n, i_n, f_n), since it will handle any neuron computation as in a sequential program. Indeed, any standard neuron computation may be performed by means of a loop that updates a variable with respect to the neuron inputs, and a final computation that maps this variable to the neuron output. θ_n stands for the initialization value. The iteration function i_n stands for the updating function inside the loop. The neuron output is finally computed with f_n. See [16] for the definition of (i_n, f_n) so as to obtain most standard neurons:

- A d-input sigmoidal neuron of a multilayer perceptron (MLP) computes

$$\sigma\left(\theta + \sum_{j=1}^{d} w_j x_j\right) \tag{1}$$

  where σ is a sigmoid function (bounded and monotonous) and θ is a threshold value.

- A d-input radial basis function of a RBF network computes $\gamma\left(\sqrt{\sum_{j=1}^{d}(w_j(x_j - t_j))^2}\right)$, where γ is a gaussian function and $\vec{t}$ is a translation vector.

- A d-input wavelet neuron of a wavelet network ([26, 3]) computes $\prod_{j=1}^{d} \psi(w_j(x_j - t_j))$, where ψ is a wavelet function (localized in both space and frequency domains).
- ... etc.

When the FPNA graph and the different operators have been defined, a general implementation can be given: each resource corresponds to a basic block, and these blocks are organized according to the graph. This implementation may be used by any FPNN derived from this FPNA. Some of these FPNNs compute very complex functions (equivalent to standard neural networks), though the FPNA graph is made simple (reduced number of edges, limited node fan-ins and fan-outs, so that the FPNA is easily mapped onto the configurable hardware).

2.2 FPNNs

A FPNN (field programmed neural network) is a FPNA which resources have been connected in a specific way. Moreover, some operator parameters must be specified.

2.2.1 Formal definition of FPNNs (and interpretation)

A FPNN is specified by means of:

- a FPNA (that is the neural resources it can use),
- for each node n in $\mathcal{N} - \mathcal{N}_i$,
  - a real value θ_n (initial value of the variable updated by the iteration function i_n)
  - a positive integer a_n (number of iterations before a neuron applies its transfer function)
  - for each p in $Pred(n)$, two real value $W_n(p)$ and $T_n(p)$ (coefficients of the affine operator $\alpha_{(p,n)}(x) = W_n(p)x + T_n(p)$),
  - for each p in $Pred(n)$, a binary value $r_n(p)$ (set to 1 iff the communication link (p, n) and the neuron in n are connected),
  - for each s in $Succ(n)$, a binary value $S_n(s)$ (set to 1 iff the neuron in n and the communication link (n, s) are connected),
  - for each p in $Pred(n)$ and each s in $Succ(n)$, a binary value $R_n(p, s)$ (set to 1 iff the communication links (p, n) and (n, s) are connected),
- for each input node n in $\mathcal{N}_i$,
  - a positive integer c_n (number of global inputs sent to this node),
  - for each s in $Succ(n)$, a binary value $S_n(s)$ (see above).

2.2.2 Computing in a FPNN

Several computation methods have been defined for the FPNNs. Their common principle may be described as follows:

- All resources behave independently.
- A resource receives values. For each value,
  - the resource applies its local operator(s),

– the result is sent to all neighbouring resources to which it is locally connected (a neuron resources waits for a_n values before sending any result to its neighbours).

The main differences with the standard neural network computation are:

- A resource may or may not be connected to a neighbouring resource. It is set by the $r_n(p)$, $S_n(s)$ and $R_n(p,s)$ configuration values.
- A communication link may directly send values to other communication links.
- A resource (even a communication link) may handle several values during a single FPNN computation process.

The *asynchronous sequential computation* method is a general FPNN computing method. This method clearly illustrates the above principles.

2.2.3 Asynchronous sequential computation

This computation method handles a list of tasks $\mathcal{L}$ that are processed according to a FIFO scheduling. Each task $[(p,n),x]$ corresponds to a value x sent on a communication link (p,n).

Initialization:

For each input node n in $\mathcal{N}_i$, c_n values $\left(x_n^{(i)}\right)_{i=1..c_n}$ are given (global inputs of the FPNN), and the corresponding tasks $[(n,s),x_n^{(i)}]$ are created for all s in $Succ(n)$ such that $S_n(s)=1$. The order of creation corresponds to a lexicographical order on (n,i,s) (with respect to the order of $\mathcal{N}$).

For each node $n \in \mathcal{N}-\mathcal{N}_i$, two local variables are used: c_n is the number of values received by the local neuron resource, whereas x_n is the value that is updated by the iteration function i_n. Initially $c_n=0$ and $x_n=\theta_n$.

Sequential processing:

While $\mathcal{L}$ is not empty

Let $[(p,n),x]$ be the first element in $\mathcal{L}$.

1. *suppress this element in $\mathcal{L}$*
2. *$x' = W_n(p)x + T_n(p)$*
3. *for all $s \in Succ(n)$ such that $(R_n(p))(s)=1$, create $[(n,s),x']$ according to the order on the successors s in $\mathcal{N}$*
4. *if $r_n(p)=1$*
   - *increment c_n and update x_n: $x_n = i_n(x_n,x')$*
   - *if $c_n = a_n$*

   (a) *$y = f_n(x_n)$*

   (b) *reset local variables ($c_n=0$, $x_n=\theta_n$)*

   (c) *for all $s \in Succ(n)$ such that $S_n(s)=1$, create $[(n,s),y]$ according to the order on the successors s in $\mathcal{N}$*

If $r_n(p)=1$, the neuron in n is said to be receiving the value of task $[(p,n),x]$.

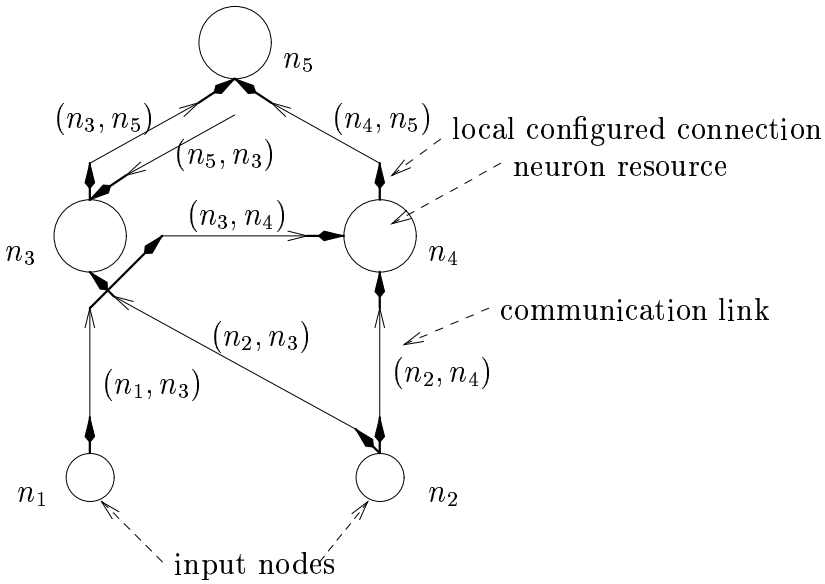

Figure 1: FPNN ϕ (resources and local connections)

2.2.4 Simple example

Let Φ be the FPNA defined by:

- $\mathcal{N} = (n_1, n_2, n_3, n_4, n_5)$
- $\mathcal{E} = \{(n_1, n_3), (n_2, n_3), (n_2, n_4), (n_3, n_4), (n_3, n_5), (n_4, n_5), (n_5, n_3)\}$
- $i_{n_3} = i_{n_4} = i_{n_5} = ((x, x') \mapsto x + x')$
- $f_{n_3} = f_{n_4} = (x \mapsto \tanh(x))$ and $f_{n_5} = (x \mapsto x)$

Figure 1 shows the neuron resources, the communication links and the configured local connections of a FPNN ϕ derived from Φ. This FPNN is more precisely defined by:

- $\theta_3 = 2.1$, $\theta_4 = -1.9$, $\theta_5 = 0$
- $a_3 = 3$, $a_4 = 2$, $a_5 = 2$
- $\forall\, (n_i, n_j) \in \mathcal{E}$ $W_{(n_i,n_j)} = i * j$ $T_{(n_i,n_j)} = i + j$
- $r_{n_3}(n_1) = 0$, $r_{n_3}(n_2) = 1$, $r_{n_3}(n_5) = 1$, $r_{n_4}(n_2) = r_{n_4}(n_3) = 1$, $r_{n_5}(n_3) = r_{n_5}(n_4) = 1$
- $S_{n_1}(n_3) = 1$, $S_{n_2}(n_3) = S_{n_2}(n_4) = 1$, $S_{n_3}(n_4) = 0$, $S_{n_3}(n_5) = 1$, $S_{n_4}(n_5) = 1$, $S_{n_5}(n_3) = 0$
- $(R_{n_3}(n_1))(n_4) = 1$, $(R_{n_3}(n_1))(n_5) = 0$, $(R_{n_3}(n_2))(n_4) = (R_{n_3}(n_2))(n_5) = 0$, $(R_{n_4}(n_2))(n_5) = (R_{n_4}(n_3))(n_5) = 0$, $(R_{n_5}(n_3))(n_3) = (R_{n_5}(n_4))(n_5) = 0$
- $c_1 = 2$, $c_2 = 1$

The asynchronous sequential computation method applies to ϕ as follows:

Initialization: Global FPNN inputs are chosen: $x^{(1)}_{n_1} = 1.5$, $x^{(2)}_{n_1} = -0.8$, $x^{(1)}_{n_2} = 1.1$. The first tasks are $[(n_1, n_3), 1.5]$, $[(n_1, n_3), -0.8]$, $[(n_2, n_3), 1.1]$ and $[(n_2, n_4), 1.1]$.

Moreover $c_{n_3} = c_{n_4} = c_{n_5} = 0$, $x_{n_3} = \theta_3 = 2.1$, $x_{n_4} = -1.9$, $x_{n_5} = 0$.

Sequential processing:

1. Task $[(n_1, n_3), 1.5]$ is first processed: $x' = 1.5\, W_{n_3}(n_1) + T_{n_3}(n_1) = 8.5$, task $[(n_3, n_4), 8.5]$ is created since $(R_{n_3}(n_1))(n_4) = 1$, and the value 8.5 is not received by the neuron in n_3 since $r_{n_3}(n_1) = 0$.
2. Task $[(n_1, n_3), -0.8]$ is processed in the same way: task $[(n_3, n_4), 1.6]$ is created.
3. Task $[(n_2, n_3), 1.1]$ is processed: $x' = 11.6$ is received by the neuron in n_3, $c_{n_3} = 1$ (therefore $c_{n_3} < a_{n_3}$), $x_{n_3} = 13.7$.
4. Task $[(n_2, n_4), 1.1]$ is processed: $x' = 14.8$ is received by the neuron in n_4, $c_{n_4} = 1$, $x_{n_4} = 12.9$.
5. Task $[(n_3, n_4), 8.5]$ is processed: $x' = 109$ is received by the neuron in n_4, $c_{n_4} = 2$, $x_{n_4} = 121.9$. Now $c_{n_4} = a_{n_4}$: reset ($c_{n_4} = 0$, $x_{n_4} = -1.9$), $y = \tanh(109) \simeq 1$, task $[(n_4, n_5), 1]$ is created.
6. Task $[(n_3, n_4), 1.6]$ is processed: $x' = 26.2$, $c_{n_4} = 1$, $x_{n_4} = 24.3$.
7. Task $[(n_4, n_5), 1]$ is processed: $x' = 29$, $c_{n_5} = 1$, $x_{n_5} = 29$.

This simple example is not useful: one would expect ϕ to map the global input values to some global output values (for example computed by n_5). Yet it shows some specific characteristics of FPNN computing: the values processed by the links towards n_3 are not always received by the neuron in n_3, some values processed by communication links are directly sent to other communication links (e.g. (n_1, n_3) towards (n_3, n_4)), some links and some local connections are not used (see (n_5, n_3)), some neurons receive values without sending outputs (n_3, n_5), etc.

2.2.5 Other general FPNN computing methods, theoretical properties

An asynchronous parallel computation method has also been defined. It is based on a request-acknowledge protocol between connected resources (neurons as weel as communication links). This parallel computation directly leads to a general implementation architecture (see [17]): predefined blocks (configurable resources) are given, they perform both resource computations and asynchronous protocol handling, and they are assembled according to the FPNA graph. Since the FPNA concept allows to handle complex aplications with simple FPNA topologies (see [16, 15] and subsection 4.3), this implementation method is flexible, straightforward and compatible with the hardware constraints.

Other computation methods (sequential as well as parallel) have been defined for recurrent FPNNs. A FPNN is recurrent when its configured local connections induce cyclical dependencies between the FPNA resources.

Theoretical aspects (halt, deadlock, determinism) have been studied for all defined computation methods ([16]). The theoretical study of FPNNs also includes the computation power of these new neural models, the optimization of weight vectors independence, or the gradient computation of feedforward deterministic FPNNs ([16, 15]).

These studies validate the FPNA/FPNN concept as a powerful and original neural framework. Next subsection gives an example based on a textbook case of neural network problems.

In a more practical way, all necessary tools and results to evaluate and handle all kinds of FPNNs are given in [16]. Moreover, it is shown how to build a hardware-friendly FPNN that computes the same global function as a given standard neural network which complexity goes far beyond the hardware constraints. Subsection 4.3 give some examples of such built FPNNs.

3 Synchronous FPNNs

A synchronous FPNN is a FPNN in which all resources may simultaneously process all its received values. In such a FPNN, any resource can "wait" for all its tasks before it begins to process them, without modifying the computation of the whole FPNN. Simplified computation methods have been defined for these synchronous FPNNs.

3.1 Definition

Several conditions must be fulfilled by a FPNN to be synchronous.

1. All iteration functions i_n must be associative and commutative.

2. Whatever computation method is chosen, the neuron in node n exactly receives a_n values. In the asynchronous sequential computation method, it means that for all n, there are exactly a_n tasks $[(p,n),x]$ where $p \in Pred(n)$. Therefore a neuron always sends an output (except if $a_n = 0$) and it does not receive any value after having sent this output. This condition is fulfilled if the following recursive property is satisfied:

$$\begin{aligned} \forall n \quad a_n &= \sum_{p \in Pred(n)} d_{(p,n)} r_n(p) \\ \text{where} \quad d_{(p,n)} &= \chi_{\mathbb{N}_+^*}(a_p) S_p(n) + \sum_{q \in Pred(p)} d_{(q,p)} (R_p(q))(n) \end{aligned}$$

3. A communication link must be able to apply a common processing to its received values, whatever neurons may directly or indirectly receive the result of this processing. Moreover this common processing must be uniformly applied by all communication links. These conditions are fulfilled if there is an associative and commutative operator $\odot$ of $\mathbb{R}$, and if there is a function $\mathcal{I} : \mathbb{R}^3 \times \mathbb{N} \to \mathbb{R}$, such that:

$$\forall\, ((p,n),n') \in \mathcal{E} \times \mathcal{N} \quad \text{and} \quad \forall\, (x^{(1)}, \ldots, x^{(d_{(p,n)})}) \in \mathbb{R}^{d_{(p,n)}} \quad \text{and} \quad \forall \text{ affine operator } \alpha$$

$$\begin{aligned} & i_{n'}(\ldots i_{n'}(\alpha(\alpha_{(p,n)}(x^{(1)})), \alpha(\alpha_{(p,n)}(x^{(2)}))) \ldots, \alpha(\alpha_{(p,n)}(x^{(d_{(p,n)})}))) \\ & \qquad = \alpha\left(\mathcal{I}\left(x^{(1)} \odot x^{(2)} \odot \ldots \odot x^{(d_{(p,n)})}, W_n(p), T_n(p), d_{(p,n)}\right)\right) \end{aligned}$$

and such that for all y:

$$\mathcal{I}\left(x^{(1)} \odot x^{(2)} \odot \ldots \odot x^{(d)}, W, T, d\right) \odot y = \mathcal{I}(x^{(1)}, W, T, 1) \odot \ldots \odot \mathcal{I}(x^{(d)}, W, T, 1) \odot y$$

This third condition is rather tricky. Nevertheless, it is satisfied by a set of easily identified FPNNs, for which all iteration functions are simple additions and for which all communication links apply linear transforms:

- $\forall\, n \in \mathcal{N} - \mathcal{N}_i \quad i_n(x,y) = x + y$
- $\forall\, n \in \mathcal{N} - \mathcal{N}_i\ \forall\, p \in Pred(n) \quad T_n(p) = 0$

Therefore, only this set of FPNNs will be considered from now on. This set includes the FPNNs defined so as to perform the computations of standard sigmoidal neurons.

3.2 Implementation properties of synchronous FPNNs

Both learning and generalization phases of synchronous FPNNs may be efficiently implemented on configurable hardware. This key feature shows that this new kind of neural models is particularly able to take advantage of the characteristics of reconfigurable computing systems: prototyping may be efficiently performed, and then the system may be reconfigured so as to implement the learned FPNN with an outstanding generalization speed (see subsection 4.3).

3.2.1 Generalization

It is shown in [16] that the computation performed by a synchronous FPNN may be simulated by a more standard neural network that belongs to the generalized model of [12]:

- each neuron resource becomes a standard sigmoidal neuron with unit weights,
- each communication link becomes a standard linear neuron,
- each configured local connection becomes a standard connection between neurons.

This result implies that each neural resource r of a synchronous FPNN may behave as a simple operator from $\mathbb{R}^{c_r}$ to $\mathbb{R}$, where c_r is the number of resources that are locally connected to r. Therefore, no protocol is required for the implemented resources to exchange data. *It leads to a very efficient pipeline implementation of the generalization phase*, see subsection 4.2.

If the following notations are used:

- y_n is the output computed by the neuron in node n
- $y_{(p,n)}$ is the output computed by communication link (p, n)

then the computations performed by a synchronous FPNN are:

- input nodes: $\forall\, n \in \mathcal{N}_i \quad y_n = \sum_{i=1}^{c_n} x_n^{(i)}$
- communication links: $\forall\, (n,s) \in \mathcal{E} \quad y_{(n,s)} = W_s(n) \left(\sum_{(R_n(p))(s)=1} y_{(p,n)} + S_n(s) y_n \right)$
- neurons: $\forall\, n \in \mathcal{N} \quad y_n = f_n \left(\sum_{r_n(p)=1} y_{(p,n)} + \theta_n \right)$

3.2.2 Learning

The learning phase of many standard neural networks (such as MLPs) is often performed thanks to what is called the back-propagation learning algorithm. This well-known algorithm is indeed the combination of the back-propagation method (for gradient computing) and of the gradient descent algorithm. Back-propagation allows an efficient computation of the gradient of an error function with respect to the parameters of a neural network. Only local computations are required. As soon as they are computed, the gradient values are used by a gradient based learning algorithm which modifies the parameters to make the neural network perform a given task.

The stochastic gradient descent is the simplest gradient based learning algorithm. For each iteration of this algorithm, a training pattern (input and expected output) is given. An error function Err estimates the distance between the neural network output and the expected output.

The gradient of this error with respect to the neural network parameters is computed. If the positive real ϵ is given as learning rate, each parameter p is updated as follows:

$$p \Leftarrow p - \epsilon \frac{\partial Err}{\partial p} \tag{2}$$

The generalized back-propagation algorithm of [12, 13] may be applied to a neural network that simulates the computation performed by a synchronous FPNN (see [16]). It shows that the gradient of a synchronous FPNN may be computed by means of simple formulae. Moreover, these computations may be performed by the resources with only local data (provided by the neighbouring resources). *It allows to define a simple hardware implementation of the learning phase*, see section 4.1. This implementation is based on assembled predefined blocks, which are similar to the blocks used in the implementation of the general asynchronous parallel computation method.

A node in a synchronous FPNN is an output node if its local neuron computes an output value without sending it to any other resource (i.e. $S_n(s) = 0$ for all $s \in Succ(n)$). Let Err be the quadratic distance $Err = \frac{1}{2}\sum_{n \text{ output node}}(y_n - e_n)^2$, where e_n is the corresponding expected output. The following notations are also introduced:

- $\partial Err_{(p,n)}$ is the differential of Err with respect to the output of communication link (p, n)
- ∂Err_n is the differential of Err with respect to the output of the neuron in node n

Then the gradient of Err with respect to the FPNN learnable parameters $W_n(p)$ and θ_n is computed thanks to the following back-propagation formulae:

- If n is an output node: $\partial Err_n = y_n - e_n$.
- If $(p, n) \in \mathcal{E}$:

$$\partial Err_{(p,n)} = r_n(p)\, \partial Err_n\, f_n' \left(\sum_{r_n(p)=1} y_{(p,n)} + \theta_n \right) + \sum_{(R_n(p))(s)=1} W_s(n) \partial Err_{(n,s)}$$

- For any neuron: $\partial Err_n = \sum_{S_n(s)=1} W_s(n) \partial Err_{(n,s)}$
- Differential w.r.t. θ_n: $\dfrac{\partial Err}{\partial \theta_n} = \partial Err_n\, f_n' \left(\sum_{r_n(p)=1} y_{(p,n)} + \theta_n \right)$
- Differential w.r.t. $W_s(n)$: $\dfrac{\partial Err}{\partial W_s(n)} = \left(\sum_{(R_n(p))(s)=1} y_{(p,n)} + S_n(s) y_n \right) \partial Err_{(n,s)}$

4 FPGA implementation of synchronous FPNNs

4.1 Learning

The implementation architectures of this subsection perform all the required computations of a stochastic gradient descent learning algorithm (see 3.2). These computations are local, so that each proposed architecture corresponds to one neural resource in the FPNA. These basic *bricks* successively handle:

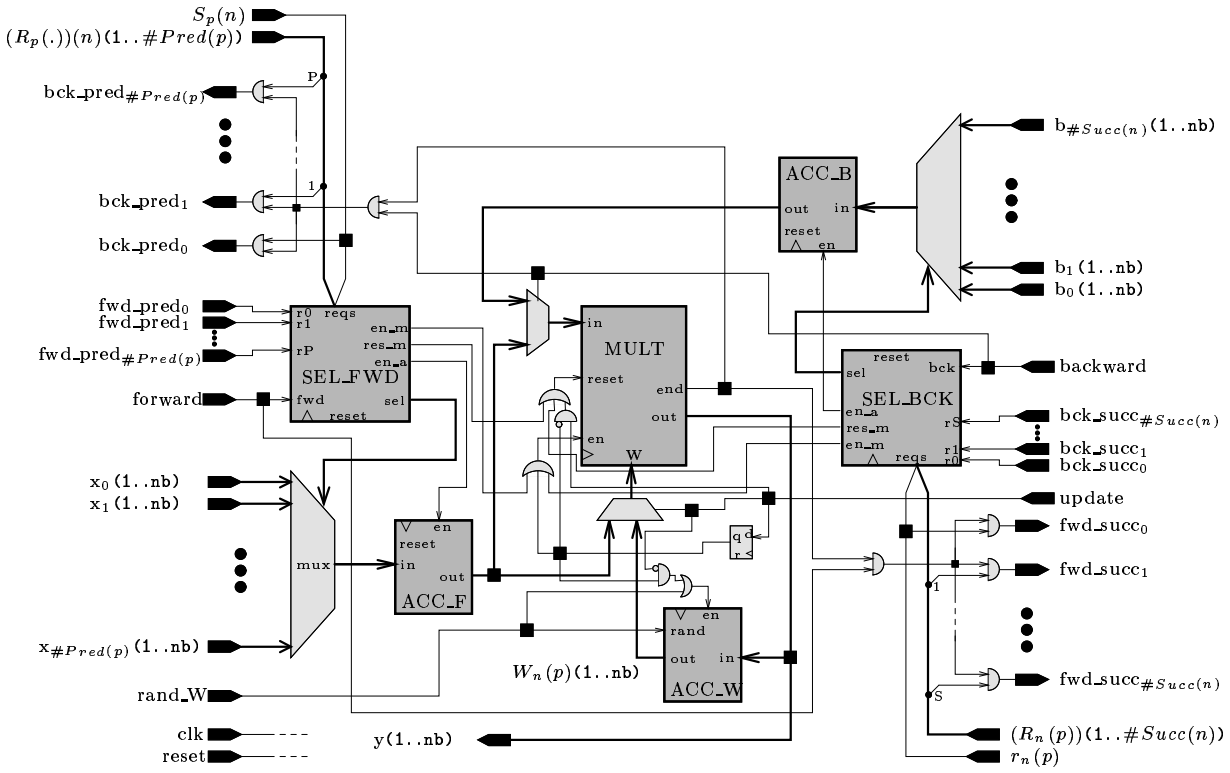

Figure 2: Communication link architecture (learning)

- the computation of the resource output
- the computations required in the backpropagation process
  - a neuron resource in n back-propagates $\partial Err_n\, f'_n\left(\sum_{r_n(p)=1} y_{(p,n)} + \theta_n\right)$ towards its connected predecessors
  - a communication link (n,s) back-propagates $W_s(n)\partial Err_{(n,s)}$
- the computations required to update the local learnable parameters.

4.1.1 Communication links

Figure 2 shows a possible implementation of a communication link with learning. All flip-flops use asynchronous reset active on '1'. The clock and reset signal connections are not shown in order to lighten this figure. The reset signal is active before each new iteration of the learning algorithm. There are six main blocks:

- `MULT` is a multiplier. It is shared by the different computation steps. Signal `end` is active when the multiplier result is available on bus `out`.

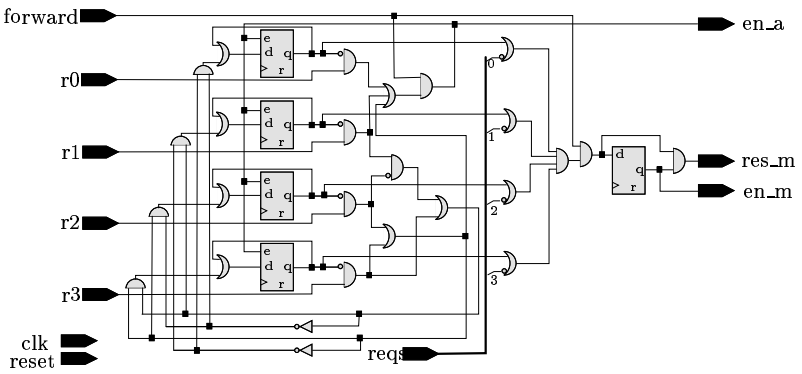

Figure 3: Blocks SEL_FWD and SEL_BCK

- ACC_F accumulates the received values. Since the FPNA/FPNN concept has been defined to ensure limited fan-ins, there is no need to use additional bits to store the accumulation result. This result stored in ACC_F keeps unchanged when the back-propagation computations are performed, that is when signal forward is idle.
- SEL_FWD receives a request signal when any connected predecessor sends a value to the communication link. This block selects a request signal, so that ACC_F accumulates the corresponding value. When all expected values have been received, SEL_FWD, signal en_m is set active, so that MULT multiplies the local weight $W_{(p,n)}$ by the accumulated value in ACC_F. When the multiplication result is available, request signals are sent to any connected successor. Figure 3 shows such a SEL_FWD block, when the resource fan-in is equal to 4.
- ACC_B and SEL_BCK are similar to ACC_F and SEL_FWD. They are used for the back-propagated computations.
- ACC_W stores the local weight $W_{(p,n)}$. When signal update is active, the values stored in ACC_F and ACC_B are multiplied. The result is then accumulated in ACC_W: the local weight is updated.

Various implementation choices are proposed for the accumulators and the multipliers in [16]. With 16-bits[1] values (fixed point, 11 fractionary bits), 2-complement accumulators, and a semi-parallel multiplier (four 8×8 multiplications sequentially handled), this communication link implementation uses less than 240 configurable logic blocks (CLBs) on a Xilinx XC4000 FPGA. Accumulations require one clock cycle, multiplications require 5 clock cycles (including reset), at 40 MHz.

4.1.2 Neurons

The main changes with respect to a communication link are linked to the computation of f_n and f'_n. When the sigmoid function $f_n(x) = \tanh(x)$ is used, then $f'_n(x) = 1 - (f_n(x))^2$. It leads to the architecture of figure 4, where a piecewise polynomial approximation of tanh is performed: table COEFFS stores the polynomial coefficients (address given by the most significant bits stored in

[1] The required precisions are precisely studied in [16]. 16 bits are sufficient for most tested synchronous FPNNs. It confirms the studies of [22, 21] despite the major differences between standard neural networks and FPNNs.

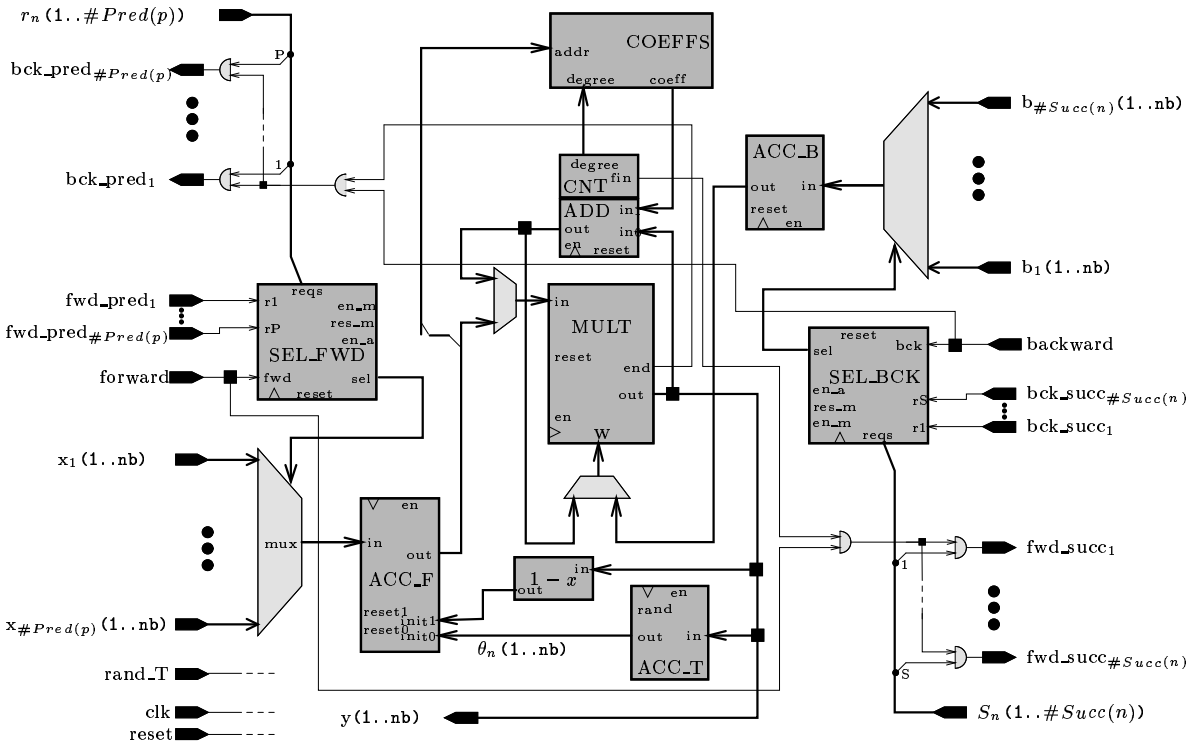

Figure 4: FPNN neuron architecture (learning)

`ACC_F`), `CNT` gives the degree of the expected coefficient, `ADD` and `MULT` compute the approximation with a Hörner scheme[2].

With the above implementation choices, less than 290 CLBs of a Xilinx XC4000 FPGA are required. The tanh function approximation lasts 11 clock cycles (a piecewise degree-2 polynomial approximation is sufficient to obtain the expected precision, provided that the coefficients are 20-bit values and `MULT` is a 16×20 semi-parallel multiplier).

4.2 Generalization

As mentioned in 3.2, the implementation of the generalization phase of a synchronous FPNN is straightforward. Each resource is a simple operator that maps its inputs to its output: one has only got to assemble these operators according to the FPNA graph, and to connect them according to the configured local connections of the FPNN.

Nevertheless, such an implementation supposes that all input values are simultaneously available for a resource. It is true if such an implementation method is applied to a regularly layered network such as a MLP. But this condition is not satisfied in most FPNNs: the different input values are generated by various numbers of resource processing. Therefore, synchronization buffers are required between connected resources, so that all input values of a resource "wait for" the latest one. Such buffers may be efficiently implemented with few CLBs (used as SRAMs which simulate shift registers, see [1]).

[2]For example $aX^2 + bX + c$ is computed as $(aX + b)X + c$.

Moreover, there is no back-propagation step here, and a synchronous FPNN is not recurrent. Therefore the successive input patterns may be processed with a pipeline: a resource locally processes its next input vector while its connected successors process its previous output. This pipeline allows to use a serial arithmetic in order to map as many resources as possible on a limited number of CLBs (area-saving operators, 1-digit width synchronization buffers). But standard serial arithmetics handle the least significant digits first: they are unable to compute tanh without interrupting the pipeline, and they first compute many useless bits in fixed point multiplications. A serial arithmetic with the most significant digits first must be chosen. Such an arithmetic is presented in [19], with a discussion about its adequation to neural computations. This *on-line* arithmetic is based on a redundant number representation system. The implementation of on-line operators on FPGAs is precisely studied in [19]. The determination of the synchronization buffer sizes is explained in [16]: it depends on the delays of the on-line operators that are successively applied so as to generate the different input values of each FPNN resource.

4.3 Applications

Standard neural benchmarks have been tested with different FPNNs (Breiman's waves of [6], Proben 1 benchmarking project of [23], etc).The benchmarking rules of [23] have been respected. These experiments attest that FPNNs may learn classifications as well as standard models (gradient descent learning algorithm), although they use an almost 2D underlying topology and 4 to 10 times less resources than the equivalent standard neural networks.

4.3.1 Example: *diabetes* classification problem

The Proben 1 project includes many different problems to be solved by neural networks in order to compare different neural models or learning algorithms. FPNNs have been applied to all real vector classification problems of the Proben 1 project in [16]. For each of these problems, an optimal shortcut perceptron has been determined in [23]. The computation power of such a neural network depends on the number of layers and on the number of neurons in each layer. FPNNs have been applied in the following way:

> A *synchronous* FPNN is defined in order to compute the same kind of global function as the standard architecture of the optimal shortcut perceptron found in [23]. This FPNN derives from a FPNA with as many neurons as in the optimal shortcut perceptron[3]. The FPNA topology must be hardware-friendly. The FPNN parameters are initialized and learned so as to solve the classification problem. The aim is to obtain a classification rate equivalent to the one of the optimal shortcut perceptron, but with a far simpler architecture allowed by the FPNA concept.

The results of the different tests are given in [16]. Several initialization methods (see [15]) and gradient-based learning algorithms have been tested: it changes the learning time (number of learning iterations), but the final performance is not significantly modified. The classification performances of the defined FPNNs are similar to the optimal classification rates obtained in [23]. The corresponding FPNAs are easy to map onto configurable hardware with the help of the implementation architectures described in subsections 4.1 and 4.2. They use 5 to 7 times less neural resources than the equivalent optimal shortcut perceptron.

[3] A multilayer perceptron consists of several ordered layers of sigmoidal neurons. Two consecutive layers are fully connected. A shortcut perceptron also consists of several ordered layers of sigmoidal neurons. But a neuron in a layer may receive the outputs of the neurons in all previous layers. A layer which is not the input layer nor the output layer is said *hidden*. A MLP is indeed a special case of shortcut perceptron, where the weights of the shortcut connections are zero (connections between non-consecutive layers).

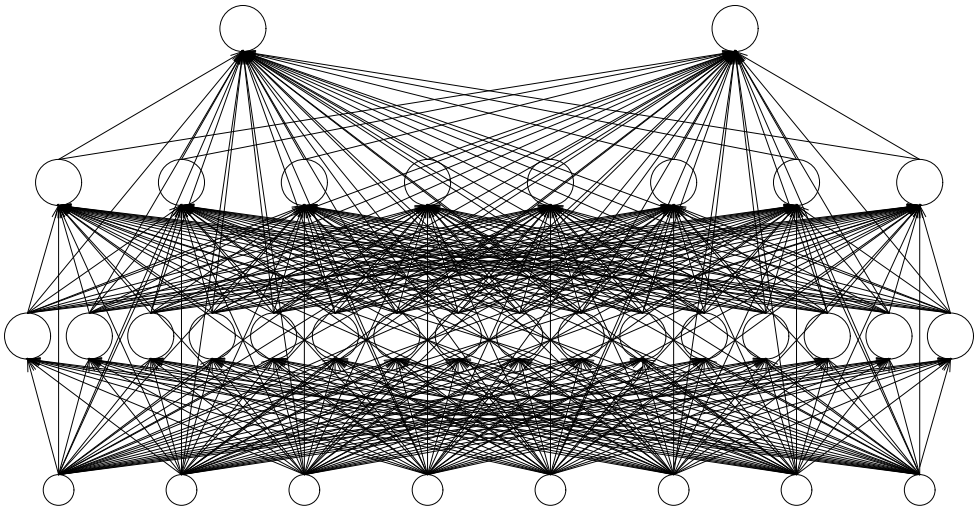

Figure 5: Optimal shortcut perceptron for the *diabetes2* problem

The *diabetes2* problem of the Proben 1 project may be taken as an example. Vectors of $\mathbb{R}^8$ must be classified into two classes (diabetes diagnosis with respect to quantified medical parameters). The optimal shortcut perceptron found in [23] uses two hidden layers of 16 and 8 sigmoidal neurons. All possible shortcut connections are used. Its average performance is 74.08 % correctly classified patterns (average on different runs of the learning process). A synchronous FPNN with 5 times less communication links reaches 73.37 % correctly classified patterns.

4.3.2 FPNN determination and implementation

The optimal shortcut perceptron for the *diabetes2* problem is shown in figure 5. The synchronous FPNN which has been determined so as to compute the same kind of global function is shown in figure 6. It also uses $16 + 8 + 2$ neurons, since the same number of non-linear operators is required so as to compute the same kind of global function. The number of communication links is greatly reduced with respect to the shortcut perceptron. This reduction is allowed by the fact that the FPNA concept shares the 78 communication links among the various combinations of successive affine operators. Each combination corresponds to one of the 384 connections of the shortcut perceptron.

The implementation of the learning phase of this FPNN requires 26500 CLBs. Four Xilinx XC40250 are used. The small fan-ins of the FPNA resources allow to exchange all required signals with the 448 available I/O ports of the FPGAs: at most 408 ports are required for any of the four FPGAs. The shortcut perceptron of figure 5 could not be directly mapped onto any number of FPGAs: these FPGAs would have to exchange too many signals, besides the topology and fan-in problems inside each FPGA. One learning iteration of the FPNN lasts 378 clock cycles, so that a 11 FPNN-MCUPS speed is reached: 11.10^6 communication links are updated by the learning process per second. It corresponds to 55 MCUPS (55.10^6 connections updated per second) with the optimal shortcut perceptron which outputs the same function as the FPNN.

The implementation of the generalization phase of this FPNN requires only one Xilinx XC40250 (7640 CLBs). It may use only 11-bit precision values. 560 CLBs are used for the synchronization buffers. The I/O bandwidth is $80\,\mathrm{MB.s^{-1}}$. A 200 FPNN-MCPS speed is reached. It corresponds to 1 GCPS (giga connections per second) with the equivalent optimal shortcut perceptron. In a

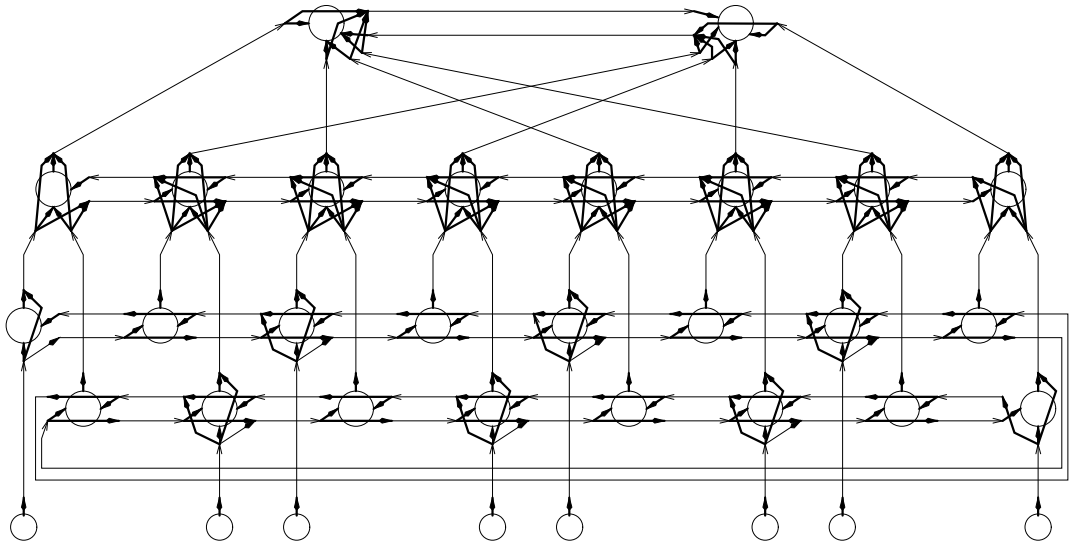

Figure 6: Equivalent FPNN architecture

reconfigurable computing system with the equivalent of the four XC40250 required for the learning phase, the patterns may be successively processed by the four FPGAs, so that a 4 GCPS equivalent speed is reached (320 MB.s^{-1} bandwidth).

These speeds must be compared with the different neural network implementations discussed in section 1. The performances of the few FPGA-based learning implementations range from 0.1 MCUPS to 4 MCUPS, whereas 2 MCPS to 30 MCPS have been reached for the generalization phase. The performances for the various FPNNs of [16] are more similar to the ones of complex neuro-computers: 5 MCUPS to 1 GCUPS and 10 MCPS to 5 GCPS (neuro-computers CNAPS, RAP, etc, cf [25]). But only one or a few FPGAs are required.

5 Conclusion

Neural network hardware implementations have to reconcile simple hardware topologies with often complex neural architectures. FPNAs have been defined for that. Their computation scheme creates numerous virtual neural connections by means of a limited set of communication links, whatever the device, the arithmetic, and the neural structure.

The general FPNA concept is first described. Then this report focuses on a particular set of *synchronous FPNNs*. Such neural networks may exploit reconfigurable computing systems in a striking way, by alternating learning and generalization phases with flexible and efficient implementations.

Nevertheless, great implementation speeds are not the main advantage of the FPNA/FPNN concept: above all, this new neural framework allows to define neural networks that are easy to map onto some configurable hardware (hardware-friendly assembling of predefined blocks), whereas such a straightforward parallel implementation is impossible for the equivalent standard neural networks.

References

[1] P. Alfke. Efficient shift-registers, LFSR counters, and long pseudo-random sequence generators. Application note XAPP 052, Xilinx, 1996.

[2] S.L. Bade and B.L. Hutchings. FPGA-based stochastic neural networks - implementation. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 189–198, 1994.

[3] R. Baron and B. Girau. Parameterized normalization : application to wavelet networks. In *Proc. IJCNN*, volume 2, pages 1433–1437, 1998.

[4] J.-L. Beuchat. Conception d'un neuroprocesseur reconfigurable proposant des algorithmes d'apprentissage et d'élagage: une première étude. In *Proc. NSI Neurosciences et Sciences de l'Ingénieur*, 1998.

[5] N.M. Botros and M. Abdul-Aziz. Hardware implementation of an artificial neural network. In *Proc. ICNN*, volume 3, pages 1252–1257, 1993.

[6] L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. *Classification and regression trees.* 0-534-98054-6. Wadsworth Inc, Belmont California, 1984.

[7] V.F. Cimpu. Hardware FPGA implementation of a neural network. In *Proc. Int. Conf. Technical Informatics*, volume 2, pages 57–68, 1996.

[8] J. Cloutier, E. Cosatto, S. Pigeon, F. Boyer, and P. Simard. VIP: an FPGA-based processor for image processing and neural networks. In *Proc. MicroNeuro*, pages 330–336, 1996.

[9] J.G. Eldredge and B.L. Hutchings. RRANN: a hardware implementation of the backpropagation algorithm using reconfigurable FPGAs. In *Proceedings of the IEEE World Conference on Computational Intelligence*, 1994.

[10] F. Elie. *Conception et réalisation d'un système utilisant des réseaux de neurones pour l'identification et la caractérisation, à bord de satellites, de signaux transitoires de type sifflement.* PhD thesis, LPCE, Université d'Orléans, 1997.

[11] A. Ferrucci. *ACME: a FPGA implementation of a self-adapting and scalable connectionist network.* PhD thesis, University of California, 1994.

[12] C. Gégout, B. Girau, and F. Rossi. A general feedforward neural network model. Technical report NC-TR-95-041, NeuroCOLT, Royal Holloway, University of London, 1995.

[13] C. Gégout, B. Girau, and F. Rossi. Generic back-propagation in arbitrary feedforward neural networks. In *Artificial Neural Nets and Genetic Algorithms – Proc. of ICANNGA*, pages 168–171. Springer-Verlag, 1995.

[14] C. Gégout, B. Girau, and F. Rossi. A mathematical model for feed-forward neural networks : theoretical description and parallel applications. Research report RR95-23, LIP-ENSL, 1995.

[15] B. Girau. Dependencies of composite connections in Field Programmable Neural Arrays. Research report, NeuroCOLT, 1999.

[16] B. Girau. *Du parallélisme des modèles connexionnistes à leur implantation parallèle.* PhD thesis, ENS Lyon, 1999.

[17] B. Girau. FPNA, FPNN: from programmable fields to topologically simplified neural networks. Research report 99.R.019, LORIA INRIA-Lorraine, 1999.

[18] B. Girau and A. Tisserand. On-line arithmetic based reprogrammable hardware implementation of multilayer perceptron back-propagation. In *Fifth international conference on Microelectronics for Neural Networks and Fuzzy Systems - MicroNeuro'96*, pages 168–175. IEEE Computer Society Press, 1996.

[19] B. Girau and A. Tisserand. MLP computing and learning on FPGA using on-line arithmetic. *Int. Journal on System Research and Information Science,* special issue on *Parallel and Distributed Systems for Neural Computing*, 1999. To be published.

[20] H. Hikawa. Frequency-based multilayer neural network with on-chip learning and enhanced neuron characteristics. *IEEE Trans. on Neural Networks*, 10(3):545–553, 1999.

[21] P.W. Hollis, J.S. Harper, and J.J. Paulos. The effects of precision constraints in a backpropagation learning algorithm. *Neural Computation*, 2:363–373, 1990.

[22] J.L. Holt and J.-N. Hwang. Finite precision error analysis of neural network hardware implementations. *IEEE Transactions on Computers*, 42(3):281–290, March 1993.

[23] L. Prechelt. Proben1 - a set of neural network benchmark problems and benchmarking rules. Technical Report 21/94, Fakultät für Informatik, Universität Karlsruhe, 1994.

[24] V. Salapura, M. Gschwind, and O. Maisch berger. A fast FPGA implementation of a general purpose neuron. In *Proc. FPL*, 1994.

[25] N. Serbedzija. Simulating artificial neural networks on parallel architectures. *Computer*, pages 56–63, 1996.

[26] Q. Zhang and A. Benveniste. Wavelet networks. *IEEE Trans. on Neural Networks*, 3(6):889–898, Nov. 1992.